Optimizing Modular Multiplication for NVIDIA's Maxwell GPUs

Niall Emmart[†], Justin Luitjens[‡], Charles Weems[†] and Cliff Woolley[‡]

[†]College of Information and Computer Sciences
140 Governors Drive, University of Massachusetts, Amherst, MA 01003, USA.
Email: nemmart@yrrid.com, weems@cs.umass.edu

[‡]NVIDIA Corporation
2701 San Tomas Expressway, Santa Clara, CA 95050, USA.
Email: {jluitjens, jwoolley}@nvidia.com

***Abstract*—In this paper we show how we were able to achieve record rates of multiple precision (MP) modular multiplication (mulmod) operations in the new NVIDIA® MP math library (XMP) on Maxwell™, NVIDIA's most recent generation of graphics processing units (GPUs). Mulmod is a key operation that is used in multiple places within the MP library, and has many real world applications, especially in cryptography, which makes it important to achieve a highly optimized implementation. Here we reveal how multiple techniques were combined to make the best use of the GPU's instructions, registers, memory, and threads. A particularly interesting algorithmic aspect, designed to work with the 16-bit hardware multipliers found in Maxwell, is the use of a two-pass process to first compute unaligned partial products, then shift the result 16 bits to the left, then compute the aligned partial products. The new algorithms are much faster than the prior, state of the art, row-oriented multiply and reduce approach, achieving speedups of 61% at 256 bits, and 117% at 512 bits, with peaks rates of 4027 million mulmod operations at 256 bits and 1081 million at 512 bits on a GTX 980Ti.**

I. Introduction

Multiprecision (MP) modular multiplication is one of the fundamental building blocks of cryptographic algorithms from key exchange (RSA, ECC, DH, ECDH) to digital signature algorithms (DSA, ECDSA) to prime generation (Miller-Rabin) and factoring (ECM). These algorithms are widely used in the modern Internet for e-commerce, secure financial transaction, secure file exchange, peer-to-peer networking, etc. However, the computational expense of doing the large number of modular multiplications required makes it attractive to offload these computations for processing on GPUs. This problem has been widely studied and there is a large body of research covering NVIDIA's earlier generations of micro architectures. However, NVIDIA's latest generation, Maxwell, requires new algorithms and techniques to achieve high performance. Here we report on a Two-Pass approach, which achieves record performance on Maxwell GPUs and are included in the beta release of NVIDIA's XMP library [16] for multiple precision arithmetic.

This material is based upon work supported by the National Science Foundation (NSF) under Award No. CCF-1217590 and NSF Award No. CCF-1525754.

We assume the reader has general familiarity with NVIDIA's GPUs and the SIMT paradigm, however, for background please see the *CUDA C Programming Guide* [13]. We also assume the reader is familiar with multiple precision arithmetic and related algorithms.

II. Literature Review

There are two common approaches to representing large numbers, positional number systems (with a fixed radix) and residue number systems (RNS). The RNS approach is attractive because all of the moduli are independent and the computation is inherently parallel, however, a review of the RNS literature [7], [10], [15] shows that the best performing implementations all use positional number systems, which is the focus of this paper. Within positional number systems, there are three important operations: multiply, square and modulo. At the small sizes of interest, the asymptotically faster algorithms such as Karatsuba, Toom-Cook and Montgomery folding show at best a modest improvement in performance. Instead, the significant performance improvements come from optimizing the simple O(n^2) algorithms and taking advantage of the specific hardware available on each generation of GPU. Squaring can be done in roughly half the time of multiplication by taking advantage of symmetry (we refer to this as fast squaring). For the modulo operation, all of the papers use Montgomery reductions [9] over other algorithms, such as classic long division or Barrett reductions [1], because of the poor performance of the division instruction on the GPU and the fewer correction steps needed for a Montgomery reduction. However, some elliptic curve cryptography (ECC) algorithms use special moduli, which allow for much faster reductions, see for example the NIST primes [12] and Solinas' Generalized Mersenne Numbers [14].

The positional number system papers generally fall into three categories: column oriented, row oriented and distributed, representing different approaches to summing the n^2 product terms (see Figure 1). These are described in detail below and we discuss the top results for each of the approaches. The column oriented and row oriented approaches assign a problem instance (either an RSA modexp, or ECC

1063-6889/16 $31.00 © 2016 IEEE
DOI 10.1109/ARITH.2016.21

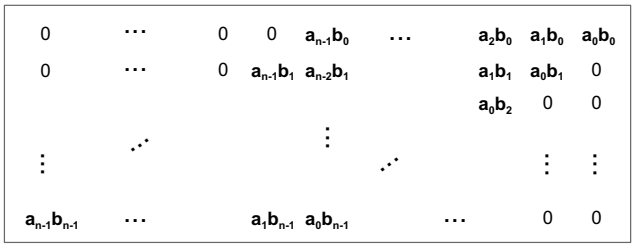

Figure 1. Product terms to be summed for an n-word by n-word multiply.

curve) to each GPU thread. The distributed approach uses a group of threads, typically in the same warp, to work collectively on a problem instance.

Column Oriented Approach: For the multiplication and Montgomery reduction operations, the column oriented approach uses an accumulator to sum down each column, from the right most column to the most left column of Figure 1 (this of course is the traditional grade school long multiplication algorithm). The main performance problem is that as each term is added, there is the possibility the accumulator will overflow, causing a carry out, which requires an extra add operation for each product term to catch the carries and is a significant waste of instruction dispatch cycles. Many papers work around this problem by using smaller limb sizes to represent A and B. For example, in [2], Bernstein et al. build a 210-bit multiplier using a sequences of 15 limbs of 14 bits each to represent A and B. The maximum column sum is $15 \cdot (2^{14}-1) \cdot (2^{14}-1)+c$, where c (the carry-in from the previous column) is less than $15 \cdot 2^{14}$. This fits in a 32-bit accumulator with no possibility for carry out. They report 461 million 210-bit mulmods/sec on a GTX 295. In [4] Emmart and Weems construct a 48-bit accumulator and use limbs of 20 to 22 bits. At 256 bits, they achieve 822K modular exponentiations per second on a GTX 260. Scaling for card, clock speed and operations, this would be equivalent to roughly 816 million 210-bit mulmods/sec on a GTX 295[†]. Using smaller limb sizes also has some overhead since the initial n-bit value must be sampled into limbs and at the end of the computation the limbs must be packed into a final result. Additionally, there is overhead at the end of each column to extract the limb value from the column sum and to compute c, the carry-in for the next column.

Row Oriented Approach: The row oriented approach takes advantage of a special feature of some NVIDIA GPUs – the multiply and accumulate with carry-in and carry-out instructions, e.g., ***madc.lo.cc*** **d, a, b, c** and ***madc.hi.cc*** **d, a, b, c**. These instructions take 32-bit values a, b, c. They compute the full 64-bit product of a*b, then extract the low or high 32 bits of the product, respectively, then add c, with carry-in and carry-out and store the result in d. All of this happens

[†]Scaling: 822K $\cdot \frac{60 \text{ SM}}{24 \text{ SM}} \cdot \frac{1241 \text{ mhz}}{1295 \text{ mhz}} \cdot \left(\frac{256 \text{ bits}}{210 \text{ bits}}\right)^2 \cdot 279$ mulmods = 816 million. The 279 mulmods is actually 74 mulmods and 260 sqrmods, but we scale the sqrmods by 208/264 because a 256-bit sqrmod requires 208 32-bit multiplies vs. 264 for a mulmod.

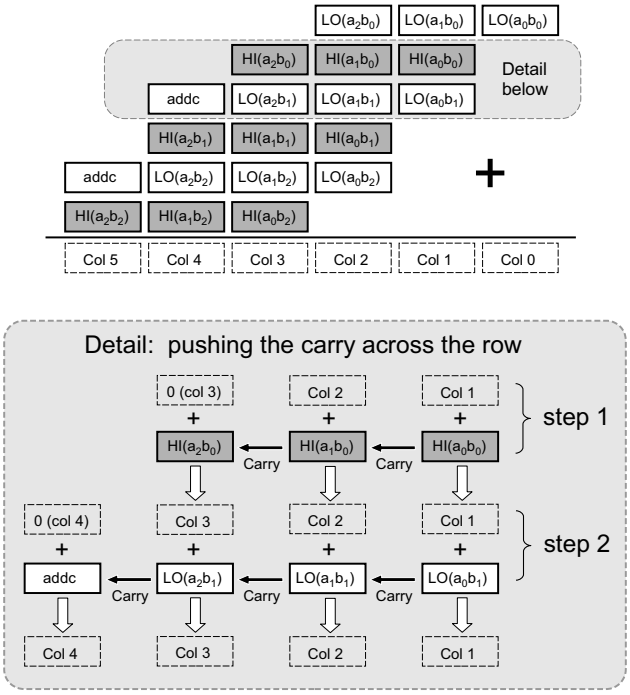

Figure 2. Example of a 3-word by 3-word multiply using the row oriented approach

with a single instruction dispatch. To build an n word by n word multiplier (or reducer), use a 32-bit accumulator for each column. Then process each row, right to left, pushing the carries along the row. For each row, first sum in all of the **lo** products then sum in all of the **hi** products, as shown in Figure 2. With this approach, most of the carries are consumed by the next multiply instruction, however, at the end of each row of **lo** products, there is a possibility for a carry out which requires a single add instruction per row. These are shown as ***addc*** operations in the figure. The row oriented approach is described well by Zheng et al. in [18] where they achieve 3.412 billion 256-bit multiplications (not mulmods) per second on a GTX Titan card. Emmart and Weems [4] also use this approach and report a throughput of 6.753 million 256-bit modular exponentiations per second on a GTX 780Ti. Scaling the results to reflect the differences in cards, clock rates and operations, Emmart and Weems achieved the equivalent of 3.469 billion 256-bit multiplications per second[‡]. The throughput results are virtually identical.

Distributed Approach: In the distributed approach, multiple threads work together to solve a single problem instance. The general idea is to divide up the columns amongst the threads. If there are more columns than threads, then assign consecutive columns to each thread. The computation is similar to the column oriented approach, except the columns are processed

[‡]Scaling: $6753K \cdot \frac{14 \text{ SM}}{15 \text{ SM}} \cdot \frac{837 \text{ mhz}}{875 \text{ mhz}} \cdot 279 \cdot \frac{264}{128} = 3.469$ billion. A 256-bit mulmod requires 264 32-bit multiplication instructions, vs. 128 for a 256-bit multiply.

in parallel. Due to the diagonal nature of Figure 1, after each row is processed, the columns are all shifted one column to the right using a warp shuffle. To minimize inter-thread communication, all the papers use a carry save approach (also known as lazy propagation), where the carries are stored locally in each thread and then resolved at the end of each multiply/reduce cycle. For more details, we refer the reader to [4], [8] or [17]. Using this approach, Jang et al. [8] achieve a throughput of 12K 2048-bit RSA decrypts per second on a GTX 580. Zheng et al. take an innovative approach in [17], where instead of using integer instructions to accumulate the columns, they use 23 bit limbs and double precision floating operations for the accumulators. They report a throughput of 39K 2048-bit RSA decrypts per second on a GTX Titan. For comparison, Emmart and Weems [4] use a row oriented approach for 2048-bit RSA and report a throughput of 41K on a GTX 580 and 62K on a GTX 780Ti. Scaling the GTX 780Ti performance to a GTX Titan results in 55K 2048-bit RSA decrypts. These results are considerably faster than those achieved with the distributed approach, which has two main drawbacks. First, the carry resolution after each multiply/reduce step requires inter-thread communication and related overhead. Second, as far as we know, no one has found an efficient way to implement fast squaring with a distributed approach.

One other paper that doesn't fit neatly in the above categories but deserves a mention is Neves and Araujo's early work implementing RSA on a GTX 260. They explored both column oriented and row oriented approaches, but they were using a first generation (1.x) graphics card, which did not support the ***madc.lo.cc*** and ***madc.hi.cc*** instructions needed to efficiently implement the row oriented approach. However, at the time, their paper set a per SM per clock performance record for the 1.x cards.

Overall, the best performance results to date, across a wide range of cards and sizes, are due to Emmart and Weems [4], except at 256-bit multiplication, where they tied with Zheng et al. [18].

The main conclusions of [4] are that at the scales which fit within registers of a single thread (roughly up to 1024-bits), the column oriented approach with the 48 bit accumulator is fastest on 1.x architectures and the row oriented approach is fastest on 2.x and 3.x architectures which has hardware instructions for ***madc.lo.cc*** and ***madc.hi.cc***. Beyond 1024-bits, the distributed approach becomes attractive because it keeps all of the data on chip by using the resources of multiple threads for a single instance.

Finally, we note the relatively poor performance observed by Emmart and Weems of the row oriented approach on the Maxwell (5.x) architecture. Their results show that on a per SM per cycle basis, the row oriented approach is roughly 33% slower on Maxwell (5.x) than on Kepler™ (3.x).

III. PTX and the Evolution of NVIDIA's Integer Multiplier

Building a high performance MP multiplier typically requires assembly language inlines to access special features of the hardware, such as the carry flag and high word of a multiplication. With NVIDIA's CUDA tool chain, this is done with Parallel Threads Execution (PTX) assembly inlines. PTX is unusual in that it is a *virtual* assembly language, where the same PTX instructions work across all generations of NVIDIA GPUs. This means that PTX instructions need not correspond one to one with the hardware instruction set of the GPU. Instead, the CUDA tool chains takes care of translating PTX to the particular ISA of the GPU running the code. Further, this means that NVIDIA can change the device instruction set, sometimes quite significantly from one generation to the next, but code that has been compiled to PTX remains upwardly compatible. Throughout the remainder of this paper, we will use bold italic fonts to signify PTX instructions and uppercase bold fonts to signify device instructions.

As discussed in the previous section, the best multiplication algorithm varies by GPU generation. For example, the column oriented approach is best for 1.x devices and the row oriented approach for 2.x and 3.x devices. The primary reason is that NVIDIA's integer multiplier at the hardware instruction level has evolved, going through three major designs, each of which requires different algorithms to fully exploit the capabilities of the hardware. This evolution is shown in Figure 3. In each

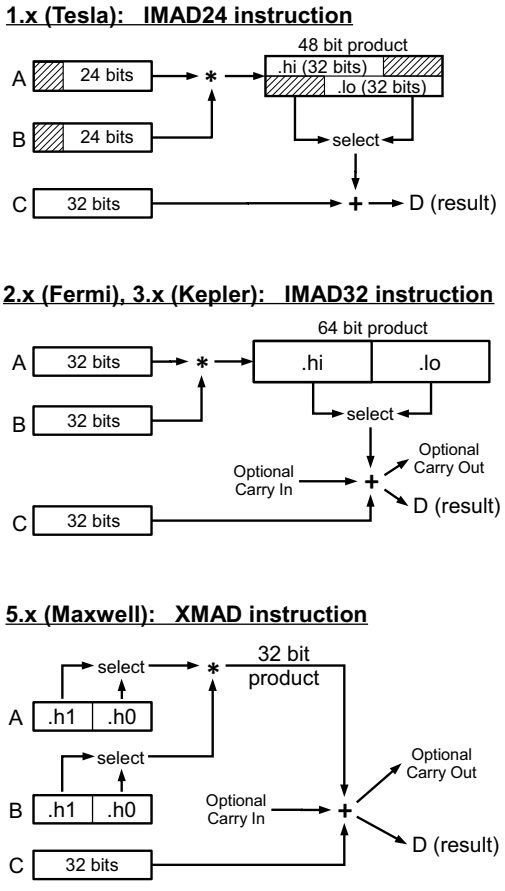

Figure 3. Evolution of the device level integer multiplier

architecture, the hardware provides a four argument multiply and accumulate instruction of the form $d \leftarrow a * b + c$, where all four registers, a, b, c and d are 32 bits in length.

The 1.x architectures multiply the least significant 24 bits of a times the least significant 24 bits of b to form a 48-bit product[§]. Then the least significant or most significant 32 bits of the product are extracted and added to c. Carry in and carry out are not supported during the add phase.

The 2.x and 3.x architectures have full 32-bit multipliers that generate a 64-bit product, and select either the low or high 32 bits of the product, and add c with an optional carry in and optional carry out.

The 5.x architectures use 16-bit multipliers. 16 bits are selected from a and 16 bits from b. These are multiplied to form a 32 bit product, which is added to c with an optional carry in and optional carry out. This instruction is called **XMAD** and has the following format:

$$\textbf{XMAD}\{\textbf{.X}\} \quad \text{d}\{\text{.cc}\},\ \text{a}\{\text{.h0}|\text{.h1}\},\ \text{b}\{\text{.h0}|\text{.h1}\},\ \text{c}$$

where .h0/.h1 select the low vs. high 16 bits, .X specifies to use carry in and .cc specifies carry out.

For 1.x and 5.x devices, the lack of a hardware 32-bit multiplier means that the standard C unsigned multiplication operation (***mul.lo***) requires multiple instructions to execute. Table I shows the number of device instructions required to compute ***mul.lo*** and ***mul.hi*** and their throughput (per cycle per SM) on the different architectures. One thing to note is that ***mul.lo*** throughput, by far the most common operation for regular C code, increases with each generation.

TABLE I
THOUGHPUT OF MUL.LO AND MUL.HI PER CYCLE PER SM

Arch	native products throughput	instructions for ***mul.lo***	***mul.lo*** throughput	instructions for ***mul.hi***	***mul.hi*** throughput
1.x	8	4	2	7	1.1
2.x	16	1	16	1	16
3.x	32	1	32	1	32
5.x	128	3	42.7	5	25.6

The 5.x architecture supports ***madc.lo.cc*** and ***madc.hi.cc*** instructions, but these are emulated with 4 and 6 native instructions respectively. Thus the row oriented approach that runs so well 2.x and 3.x requires roughly $10n^2$ native instruction on 5.x.

Consider computing the product of two 32-bit values, a and b each in registers, using a 16-bit multiplier. The result will be 64 bits and should occupy two 32-bit registers. One would first break a and b into halves, a_l, a_h and b_l, b_h and compute:

$$a_l \cdot b_l + ((a_l \cdot b_h + a_h \cdot b_l) << 16) + (a_h \cdot b_h << 32)$$

the products of $a_l \cdot b_l$ and $a_h \cdot b_h$ are efficient because they are 32-bit aligned, and thus aligned with the result registers. However the $a_l \cdot b_h$ and $a_h \cdot b_l$ products are 16-bit aligned, and thus straddle the result registers. It is these alignment problems that push the instruction count from 4 16-bit multiplies to

[§]The 1.x architectures also support a 16-bit multiplication mode, similar to the 5.x architectures.

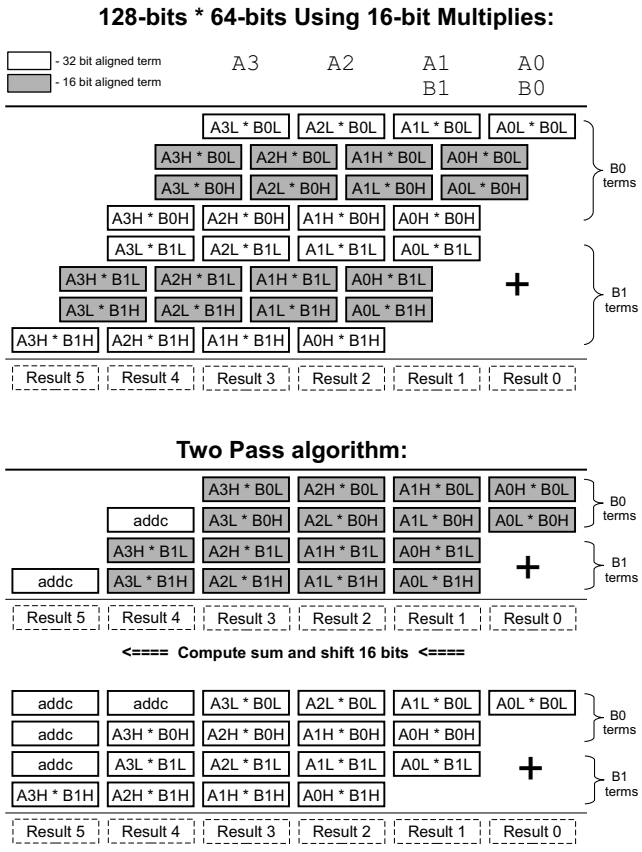

Figure 4. The Two-Pass approach to multiplication

the 10 native instructions, noted above, that are needed to perform a pair of ***madc.lo.cc*** and ***madc.hi.cc*** instructions. Thus alignment issues with Maxwell's 16-bit multiplier are the source of the poor performance of the row oriented approach noted by Emmart and Weems in [4].

IV. TWO-PASS ALGORITHMS FOR MULTIPLICATION AND MONTGOMERY REDUCTION

We begin with Figure 4. The top half of the figure shows the 16-bit products needed to compute a 4 word (128-bit) value multiplied by a 2 word (64-bit) value. Half of the product terms are 16-bit aligned (the shaded terms) and half are 32-bit aligned. The problem is the 16-bit aligned terms - the compiler has a tendency to handle them individually, and spends a lot of instructions splitting the terms and adding them to the correct result register. What's needed is an efficient way to handle all of the 16-bit aligned terms at the same time. It turns out the solution is quite simple. Just regroup all of the terms and run the computation in two phases, as shown in the bottom half of the figure. In the first phase, sum all of the 16-bit aligned terms (the shaded terms), using a row oriented approach and pushing the carry along the row, as in Figure 2. At the end of the second and fourth rows, there is the possibility of a carry out, so add instructions are needed to pick up potential carry

outs.

Once the 16-bit aligned terms have all been summed, shift all the columns 16 bits to the left. This can be done very efficiently with byte permute instructions. Now the shaded terms are all aligned correctly, and we can add in the 32-bit aligned terms. Note again, the extra add operations at the ends of the rows. Once these terms have been added, we're done.

For an n-word by n-word multiply, if the adds are done with care, this algorithm can be implemented with $4n^2$ multiply instructions, $2n$ permute instructions and $2n-2$ add instructions.

The key to implementing this algorithm is the ability to generate Maxwell **XMAD** instructions from CUDA C. CUDA allows the inlining of PTX, but there is no single PTX instruction that corresponds to the semantics of Maxwell's **XMAD**, and as of CUDA 7.5, there is no way to generate them. However, NVIDIA has provided us with an experimental early release version of the compiler which detects special PTX sequences and converts them to **XMAD** instructions on Maxwell. The PTX sequences are described in the Appendix, and as of CUDA 8.0 these will be supported in the standard compiler.

For the pseudo-code that follows, we introduce some conventions to make the code short and concise. Uppercase letters will be used to refer to multiple precision (MP) values. Lowercase letters are 32-bit values. We allow basic arithmetic and bit shifting on MP values. MP values also support word indexing, where 0 is the least significant word. We introduce four functions: LL(a, X), LH(a, X), HL(a, X), HH(a, X), used to generate the rows shown in Figure 4. The first argument is a 32-bit value and the second argument can be either an MP value or a 32-bit value. Conceptually, LH(a, X) could be implemented as follows:

```
MP LH(uint32 a, MP X) {
  // LH: build a row of low(a)*high(X[i])
  // X is w words in length

  MP ROW[w];

  for i=0 to w-1 do
    ROW[i]=low(a) * high(X[i])    // hence the name LH
  return ROW;
}

uint32 LH(uint32 a, uint32 x) {
  // scalar version
  return low(a) * high(x)
}
```

Figure 5. LH(a, X) function

where low(x) and high(x) simply return the least significant or most significant 16 bits of x respectively.

The pseudo-code for the Two-Pass multiply algorithm is given in Figure 6. The actual implementation of the Two-Pass algorithm is much longer and more complicated. It uses fully unrolled loops and most of the computation is done with inline PTX, and of course it uses chains of **XMADs** with carry in and carry out to compute the rows, in place of the LL/LH/HL/HH functions. Conceptually though, it closely follows the pseudo-code.

```
MP multiply(MP A, MP B) {
  // A and B are w words in length
  // P is 2w words in length

  MP P[2*w];

  P=0
  for i=0 to w-1 do
    P=P + (LH(B[i], A)<<32*i)
    P=P + (HL(B[i], A)<<32*i)
  end
  P=P<<16
  for i=0 to w-1 do
    P=P + (LL(B[i], A)<<32*i)
    P=P + (HH(B[i], A)<<32*i+32)
  end
  return P
}
```

Figure 6. Two-Pass multiply algorithm

```
MP reduce(MP X, MP N, uint16 np0) {
  // reduces X by the modulus N.  N is w words in
  // length, X is 2w words in length.  np0 is the
  // inverse needed for the Montgomery reductions,
  // i.e., (2^16-LL(np0, N[0])) mod 2^16 is 1.

  MP U[w];

  U=0
  for i=0 to w-1 do
    q=LL(X[0], np0);

    X=X + LL(q, N)
    U=U + LH(q, N) + (X[0]>>16)
    X=X>>32

    q=LL(U[0], np0)

    U=U + LL(q, N)
    X=X + LH(q, N) + (U[0]>>16)
    U=U>>32
  end
  X=X + (U<<16)

  if(X>=2^(32*w))
    X=X-N

  return X
}
```

Figure 7. Two-Pass Montgomery reduction algorithm

For brevity we omit the Two-Pass fast squaring algorithm. It is easily derived from the row oriented fast squaring algorithm.

In the classic Montgomery reduction of X, each iteration of the loop updates $X = X + q * N$ with q chosen such that the least significant word of the updated X will be zero. However, even if we limit q to 16 bits, the product of $q * N$ will have terms that are not 32-bit aligned, namely LH(q, N). The solution is to keep two MP values, one to store the 16-bit aligned terms (U) and another to store the 32-bit aligned terms (A), and use the following update rules:

$$U = U + \text{LH}(q, N) \quad \text{and} \quad A = A + \text{LL}(q, N)$$

Then, at the end of any given iteration, the X from the Montgomery reduction could be reconstructed with $X = X + A + (U << 16)$. However, since X and A are both 32-bit aligned, we need not keep a separate A value. We can simply add the 32-bit aligned terms to X, and the update

rules are:

$$U = U + \text{LH}(q, N) \quad \text{and} \quad X = X + \text{LL}(q, N)$$

this is the basis for the Two-Pass Montgomery algorithm and is presented in Figure 7.

V. Experimental Setup and Results

To evaluate our algorithms, we use two NVIDIA GeForce GPU cards, a GTX 780Ti and a GTX 980Ti hosted on an ASRock X79 Extreme11, configured with a quad core Intel Core i7-4820K, clocked at 3.7 GHz with 16GB of main memory running 64-bit Ubuntu Server (version 14.04.1), version 352.39 of the NVIDIA driver and an early release version of the nvcc compiler which supports the PTX sequences for XMAD generation. We use GMP version 6.0.0a with GCC version 4.8.2 to verify the results from the GPU.

Our test procedure is straightforward. We generate a large number of instances, each instance consists of a random multiple precision value, X, that is k bits in length. We also generate two random values N (the modulus) and Y (for products) each is k bits in length. The same N and Y values are used across all instances. There are no restrictions on the values of X or Y, but we ensure that N is odd and the most significant word is non-zero. To measure performance on the GPU we perform repeated squaring or multiplication (depending on the selected operation) and after each multiply/square step, we use a Montgomery reduction to reduce the $2k$ bits of the product back down to k bits. The pseudo-code for the GPU kernel is as follows:

```
MP CUR[k/32];

// load N and Y from global memory
// store N and Y in shared memory for use in the loop

// load X from global memory
CUR=X;

for i=1 to 1000 do
  if(operation==SQUARING)
    CUR=CUR * CUR;        // k-bit squaring
  else if(operation==MULTIPLICATION) {
    // load Y from shared memory
    CUR=CUR * Y;          // k-bit multiplication
  }

  // load N from shared memory
  CUR=CUR * R^-1 mod N;   // Montgomery reduction, R=2^k
end

// store CUR in global memory as the final result
```

Figure 8. GPU kernel pseudo-code

Note, that in the main loop of the kernel, we do not load any values from global memory. They always come from shared memory. This ensures that we are accurately measuring the compute performance in a way that is not influenced by global memory latency or bandwidth issues. To increase the accuracy we also use a larger loop count (10000 vs 1000) for 128-bit operations.

To generate good timing results, we run the GPU kernel 12 times on the same data. 2 runs are for warm-up and 10 runs are for timing. We average the 10 timing runs, which we report in our results. After the timing runs are complete, we copy the results from the GPU back to the CPU and verify they are correct with GMP. The run times only include the GPU compute time. They do not include the time to generate the data, copy the data to or from the GPU, or any CPU time used to verify the results. We also note that the performance is somewhat dependent on the temperature of the card, and thus we ensure the temperature (as reported by **nvidia-smi**) is less than 50 degrees C prior to each run.

TABLE II
Performance of Multiply and Reduce across four sizes

Card	Arch	Alg	Throughput & Threads per Block			
			k=128	k=256	k=384	k=512
GTX 780Ti	3.5	Row Oriented	**7922 M** 1024	**2030 M** 768	**892 M** 1024	**502 M** 768
GTX 980Ti	5.2	Row Oriented	**9227 M** 1024	**2501 M** 1024	**1131 M** 1024	**498 M** 768
GTX 980Ti	5.2	Two Pass	**12219 M** 768	**4027 M** 1024	**1948 M** 896	**1081 M** 768

TABLE III
Performance of Square and Reduce across four sizes

Card	Arch	Alg	Throughput & Threads per Block			
			k=128	k=256	k=384	k=512
GTX 780Ti	3.5	Row Oriented	**9510 M** 1024	**2602 M** 1024	**1154 M** 1024	**665 M** 768
GTX 980Ti	5.2	Row Oriented	**10862 M** 768	**3094 M** 1024	**1442 M** 1024	**648 M** 768
GTX 980Ti	5.2	Two Pass	**13680 M** 768	**4759 M** 1024	**2369 M** 896	**1326 M** 768

Table II shows the performance of repeated multiply and reduce on the two cards and across a range of sizes, showing the performance of the original row oriented approach on the 780Ti and 980Ti and the new Two-Pass algorithms on the 980Ti. We see that in all cases, the new Two-Pass algorithm is faster than the row oriented approach on the 980Ti, by 32% at 128 bits, 61% at 256 bits, 72% at 384 bits and 117% at 512 bits. For the 512 bit runs, we compile with -maxrregcount=80 to ensure that we have at least 24 warps on each Streaming Multiprocessor (SM).

In Table II, we also list the threads per block used to achieve the performance result. The number of instances run is determined by the following equation, which is designed to minimize the tail effect:

$$\text{instances} = \text{SMs} \cdot 32 \cdot \text{threads\_per\_block}$$

where SMs is 16 for the 780Ti and 22 for the 980Ti.

Table III shows the performance of repeated square and reduce with the same cards, sizes and algorithms. As with the multiply and reduce, the new Two-Pass algorithms significantly outperform the row oriented algorithms on the 980Ti. The speed ups are 26%, 54%, 64% and 105% respectively, which is only slightly less than the speedups achieved on multiply and reduce.

Table IV shows the instructions per cycle (IPC), as measured by **nvprof** for the Two-Pass algorithms running Multiply/Re-

TABLE IV
NVPROF MEASUREMENT OF IPC FOR THE TWO-PASS MULMOD AND SQRMOD OPERATIONS FROM TABLES II AND III

Card	Operation	Instructions Per Cycle & Utilization			
		k=128	k=256	k=384	k=512
GTX 980Ti	Multiply/Reduce	**3.79** 95%	**3.81** 95%	**3.71** 93%	**3.47** 87%
GTX 980Ti	Square/Reduce	**3.84** 96%	**3.85** 96%	**3.77** 94%	**3.51** 88%

TABLE V
THROUGHPUTS, NORMALIZED ON A PER SM PER MHZ BASIS

Multiply/Reduce throughput per SM per MHz

Card	Arch	Alg	k=128	k=256	k=384	k=512
GTX 780Ti	3.5	R O	603 K	154 K	68.0 K	38.2 K
GTX 980Ti	5.2	R O	390 K	106 K	47.8 K	21.0 K
GTX 980Ti	5.2	Two-Pass	516 K	170 K	82.3 K	45.7 K

Square/Reduce throughput per SM per MHz

Card	Arch	Alg	k=128	k=256	k=384	k=512
GTX 780Ti	3.5	R O	725 K	198 K	88.8 K	50.7 K
GTX 980Ti	5.2	R O	459 K	131 K	60.9 K	27.4 K
GTX 980Ti	5.2	Two-Pass	578 K	201 K	100.1 K	56.0 K

duce and Square/Reduce on the 980Ti, which correspond to the last row of Tables II and III. For code that consists entirely of integer operations, the theoretical maximum IPC on Maxwell is 4 and thus IPC/4 provides a useful utilization metric. We see that for 128, 256 and 384 bit operations, the IPC ranges from 3.71 to 3.85, with a corresponding utilization between 93% and 96%. At 512 bits, the IPC falls off slightly to roughly 3.5 (87% utilization). The fall off at 512 bits is caused by the number of registers per thread, in this case 80, which means there are not enough warps on each SM to hide all the latencies. In general, the high IPC and utilization rates tell us that the Two-Pass algorithms run very efficiently on the Maxwell hardware.

Table V shows the Multiply/Reduce and Square/Reduce performance normalized on a per SM per MHz basis. In effect this allows us to compare how efficient Kepler and Maxwell are at modular multiplication. With the traditional row oriented algorithm, we see that Maxwell is clearly much less efficient. Emmart and Weems [4] used the row oriented algorithms and this explains the poor performance they observed. However, even with the new Two-Pass algorithms, Maxwell is only a bit faster. Maxwell has a throughput of 128 16-bit multiplies per cycle and given that it takes 4 16-bit multiplies to do a full 32-bit product (with a 64-bit result), Maxwell should be able to approach 32 full products per cycle, whereas Kepler has a throughput of only 16 full products per cycle (one cycle for mul.lo and one cycle for mul.hi). An interesting question is why aren't we seeing better performance on Maxwell?

It's a three part answer. First, on Kepler, there are 4 warp schedulers, but only 1 warp can issue a mul.lo or mul.hi instruction per cycle. However, the other warp schedulers can issue non multiply instructions. This means Kepler can effectively overlap multiplies with other instructions. On multiplication dense codes such as this, the run time is dominated by just the time to run all the multiplies and all other instructions (including integer adds) get overlapped and are essentially free. This is not the case on Maxwell, where all integer instructions must compete for dispatch cycles. Thus every add, shift, permute, and move instruction executed takes dispatch slots that could contribute to the 128 16-bit products per cycle, greatly reducing the delivered multiplication throughput.

Second (and related), a significant portion of the instructions executed by the Two-Pass algorithms are not multiplies, which is especially true at the small sizes. Hence we see that at 128 bits, Kepler is more efficient (per SM per cycle) however, as the size increases, Maxwell is a bit more efficient.

Finally, as mentioned earlier, at 512 bits, we require 80 registers per thread which means there are not enough warps per SM to hide all the latencies and the IPC suffers.

Even if the per SM per cycle comparison between Maxwell and Kepler shows only modest improvement, it is still very advantageous to run modular multiplication on Maxwell because Maxwell is clocked at a higher rate, supports more SMs per card and runs at lower power. Thus we see a significant boost comparing Maxwell cards to Kepler cards.

VI. CONCLUSION

The previous work of Emmart and Weems [4], established that the row oriented approach for MP mulmod is very efficient on Fermi (2.x) and Kepler (3.x), but they found that the performance was disappointing on Maxwell (5.x). We show that the reason is that Maxwell uses a 16-bit hardware multiplier and the translation of 32-bit instructions to 16-bit native multiplies results in many wasted instructions to handle re-alignment of the results at the 32-bit size. We thus proposed new algorithms for MP multiplication, squaring, and Montgomery reductions based on a Two-Pass approach across the full width of the operand. In the first pass, we handle all the 16-bit aligned products and then we handle the 32-bit aligned products on the second pass. As a result, we were able to achieve 12.22 billion 128-bit mulmods per second and 13.68 billion 128-bit square and reduce operations per second on a GTX 980Ti, with proportional speeds at 256, 384, and 512 bits. These algorithms require the ability to generate Maxwell native 16-bit multiplies from CUDA C, which NVIDIA has made possible via an early release of the compiler. These new compiler features will be included in CUDA 8.0. We tested the new algorithms on a GTX 980Ti and found that they significantly improved the performance, especially at the larger sizes, as compared to the row oriented algorithms. High performance at these sizes is very important because they are the building blocks for even larger sizes that are relevant to RSA and other cryptographic operations. Our algorithms are included in the beta release of NVIDIA's XMP library [16].

APPENDIX
GENERATING MAXWELL XMAD INSTRUCTION

As of CUDA 8.0, the NVIDIA tool chain will look for certain special sequences of PTX that mimic the behavior of

```
// PTX sequence for:  XMAD.X  D.CC, A.H0, B.H0, C
{
    .reg .u16     %ah0, %ah1, %bh0, %bh1;
    .reg .u32     %temp;

    mov.b32       {%ah0, %ah1}, A;
    mov.b32       {%bh0, %bh1}, B;
    mul.wide.u16  %temp, %ah0, %bh0;    // A.H0 * B.H0
    addc.cc.u32   D, C, %temp;          // .X and .CC
}
```

Figure 9. PTX sequence to generate: **XMAD.X D.CC, A.H0, B.H0, C**

```
__device__ __forceinline__ void
XMADLLXCC(uint32_t& d, uint32_t a, uint32_t b, uint32_t c)
{
  asm volatile ("{\n\t"
                ".reg .u16    %ah0, %ah1, %bh0, %bh1;\n\t"
                "mov.b32      {%ah0, %ah1}, %1;\n\t"
                "mov.b32      {%bh0, %bh1}, %2;\n\t"
                "mul.wide.u16 %0, %ah0, %bh0;\n\t"
                "addc.cc.u32  %0, %0, %3;\n\t"
                "}" : "=r"(d) : "r"(a), "r"(b), "r"(c));
}
```

Figure 10. CUDA C function for: **XMAD.X D.CC, A.H0, B.H0, C**

Maxwell's XMAD instruction and it will replace the sequence with a single XMAD instruction. For example:

XMAD.X D.CC, A.H0, B.H0, C

can be implemented with the four instructions shown in Figure 9 and Figure 10 shows the corresponding CUDA device function. The two ***mov.b32*** instructions unpack the A and B into their respective low and high half words. The ***mul.wide.u16*** instruction computes the product of the two half words into a 32-bit temporary register and the final ***addc.cc.u32*** instruction adds C to the temporary, with carry in and carry out and stores the result in D. The four combinations of {A.H0, B.H0}, {A.H1, B.H0}, {A.H0, B.H1}, and {A.H1, B.H1} can be generated by changing the arguments of the ***mul.wide.u16*** instruction and the four combinations of carry in and carry out can be generated using different add instructions: ***add.u32***, ***addc.u32***, ***add.cc.u32***, and ***addc.cc.u32***.

REFERENCES

[1] P. Barrett. "Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor." In *Advances in cryptology: CRYPTO '86: proceedings*, vol. 263 of *Lecture Notes in Computer Science*, pp. 311-323, Springer-Verlag, 1987.

[2] D. J. Bernstein, H. C. Chen, M. S. Chen, C. M. Cheng, C. H. Hsiao, T. Lange, Z. C. Lin, and B. Y. Yang. "The billion-mulmod-per-second PC." in *Proceedings 4th Workshop on Special-purpose Hardware for Attacking Cryptograhic Systems*, September 2009, pp. 131-144.

[3] A. E. Cohen and K. K. Parhi. "GPU Accelerated Elliptic Curve Cryptography in GF(2m)" in *Proceedings of the 2010 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Seattle, WA, August 2010, pp. 57-60.

[4] N. Emmart and C. Weems ."Pushing the Performance Envelope of Modular Exponentiation Across Multiple Generations of GPUs" in *2015 IEEE International, Parallel and Distributed Processing Symposium (IPDPS)*, Bengaluru, India, May 2015.

[5] S. Fleissner. "GPU-Accelerated Montgomery Exponentiation." in *ICCS 2007: Proceedings of the 7th international conference on Computational Science, Part I*, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 213-220.

[6] P. Giorgi, T. Izard, and A. Tisserand. "Comparison of Modular Arithmetic Algorithms on GPUs." in *Proc. International Conference on Parallel Computing ParCo*, Lyon, France, September 2009.

[7] O. Harrison and J. Waldron. "Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware." in *Second International Conference on Cryptology in Africa*, Gammarth, Tunisia, June 21-25, 2009.

[8] K. Jang, S. Han, S. Han, S. Moon, K. Park. "SSLShader: Cheap SSL Acceleration with Commodity Processors." in *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI)*, Mar. 2011.

[9] P. Montgomery. "Modular Multiplication Without Trial Division" *Mathematics of Computation*, vol. 44, no. 170, 519-521, 1985.

[10] A. Moss, D. Page, and N. Smart. "Toward Acceleration of RSA Using 3D Graphics Hardware." in *Cryptography and Coding*, pp. 364-383, 2007.

[11] S. Neves and F. Araujo. "On the performance of GPU public-key cryptography." in *2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, IEEE, 2011.

[12] National Institute of Standards and Technology (NIST). "Recommended Elliptic Curves for Federal Government Use" [Online] Available: http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf

[13] NVIDIA Corp. "CUDA C Programming Guide" [Online] Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[14] J. Solinas. "Generalized Mersenne numbers". Technical Reports, CACR, Waterloo (1999)

[15] R. Szerwinski and T. Güneysu. "Exploiting the Power of GPUs for Asymmetric Cryptograph" in *Cryptographic Hardware and Embedded Systems (CHES)*, 2008, pp. 79-99.

[16] NVIDIA Corp. XMP Multiply Precision Library. [Online] Available: http://nvlabs.github.io/xmp/

[17] F. Zheng, W. Pan, J. Lin, J. Jing and Y. Zhao. "Exploiting the Floating-Point Computing Power of GPUs for RSA" in *Information Security: 17th International Conference (ISC)*, Hong Kong, China, Oct 12-14, 2014.

[18] F. Zheng, W. Pan, J. Lin, J. Jing and Y. Zhao. "Exploiting the Potential of GPUs for Modular Multiplication in ECC" *Information Security Applications*, Springer International Publishing, pp. 295-306, 2014.